TP14 : Reconnaissance de chiffres

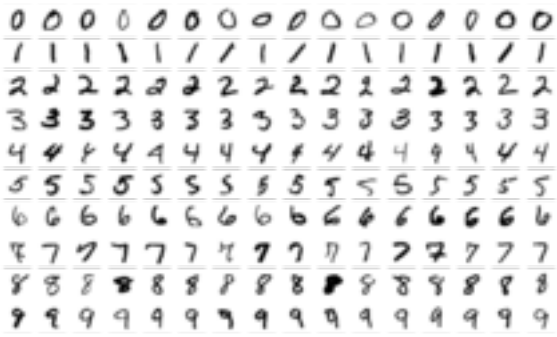
Objectifs du TP :

- Réviser la manipulation de fichiers en Ocaml et les fonctions des modules Array et List.
- Implémenter l'algorithme des k plus proches voisins.
- Manipuler sur un exemple concret les concepts relatifs à la classification supervisée.

Dans ce TP, on s'intéresse au problème de classification suivante : étant donnée une image, dire quel chiffre entre 0 et 9 elle représente. Les images manipulées seront des carrés de 28 pixels de côté, chaque pixel représentant une nuance de gris, entre 0 (blanc) et 255 (noir). Une image est donc un vecteur à $28 \times 28 = 784$ coordonnées (représentant les lignes mises bout à bout) ayant une valeur comprise entre 0 et 255 : c'est à partir de ce vecteur que notre fonction de classification devra décider la classe de l'image.

Partie 1 *Lecture des données*

Comme pour toute tâche de classification supervisée, il faut d'abord nous doter d'un ensemble d'apprentissage constitué d'objets pour lesquels on connaît déjà la classe. Ici, cet ensemble est fourni par la base de données MNIST : elle répertorie des images de chiffres manuscrits comme ceux ci-dessous pour lesquels on sait quel chiffre est censé être représenté.



Plus précisément, on travaillera dans ce TP sur :

- Un jeu d'entraînement d'environ 10000 images (la base MNIST en fournit 60000).
- Un jeu de test d'environ 1600 images (la base MNIST en fournit 10000).

Les données en question ont déjà été prétraitées de la façon suivante :

- Le fichier x_train.csv (respectivement x_test.csv) contient les images du jeu d'entraînement (respectivement de test) au format suivant : chaque ligne correspond à une image et sur une ligne se trouvent les valeurs des pixels de l'image séparées par des virgules.
- Le fichier y_train.csv (respectivement y_test.csv) contient les classes de chaque image du jeu d'entraînement (respectivement de test) au format suivant : la ligne i contient un entier entre 0 et 9 indiquant la classe de l'image en ligne i dans x_train.csv (respectivement x_test.csv).

L'objectif de cette partie est de lire ces fichiers de sorte à construire des jeux d'exemples (d'entraînement ou de test) dont le type est (image*int) array où le type image est défini par :

```
type image = int array
```

Un jeu d'exemples est donc un tableau contenant en case i un couple constitué d'une image représentée par un tableau d'entiers correspondant à ses pixels et de la classe de cette image, c'est-à-dire de l'entier représenté sur l'image. Pour répondre aux questions suivantes, on pourra relire le fonctionnement des fonctions open_in, close_in et input_line.

1. Ecrire une fonction lire_lignes_fichier de signature in_channel -> string list prenant en entrée un flux d'entrée et renvoyant une liste de chaînes de caractères dont le i-ème élément est la chaîne correspondant à la i-ème ligne sur le flux entrant.

2. Ecrire une fonction parser_ligne_image de signature string -> int array prenant en entrée une chaîne de caractères correspondant à la description d'une image et renvoyant un tableau d'entiers correspondant aux valeurs des pixels de cette image.

   Pour cette fonction, on pourra consulter la documentation au sujet de String.split_on_char *et EXCEPTIONNELLEMENT faire une conversion d'une liste vers un tableau à l'aide de* Array.of_list.

3. Ecrire une fonction lire_images de signature string -> image array prenant en entrée un nom de fichier contenant des images au format décrit par l'énoncé et renvoyant un tableau contenant en case i la représentation en Ocaml de la i-ème image.

4. Ecrire de même une fonction lire_etiquettes de signature string -> int array prenant en entrée un nom de fichier contenant des étiquettes et renvoyant un tableau contenant la i-ème étiquette en case i.

5. Ecrire une fonction jeu_donnees de signature image array -> int array -> (image*int) array prenant en entrée un tableau d'images et le tableau d'étiquettes correspondantes et renvoyant le tableau associant chaque image à son étiquette.

Afin de visualiser graphiquement les exemples dont on dispose, on construit une fonction permettant d'afficher de façon rudimentaire une image.

6. Ecrire une fonction affiche_image de signature image -> unit prenant en entrée un tableau d'entiers représentant une image et l'affichant. Pour chaque pixel, il faudra décider quel symbole dessiner en fonction de sa valeur. On rappelle que les images sont de taille 28×28 pixels.

7. A l'aide des fonctions précédentes, afficher quelques uns des exemples (image, étiquette) du jeu d'apprentissage fourni.

Bonus : Les données MNIST sont accessibles à cette adresse : http://yann.lecun.com/exdb/mnist/. Les récupérer et reprendre cette partie en allant directement lire les données dans les fichiers bruts de la base MNIST. Le format utilisé pour encoder ces fichiers est spécifié à l'adresse ci-dessus.

Partie 2 *Méthode des k plus proches voisins*

Le principe de la méthode des k plus proches voisins est le suivant : pour déterminer la classe d'un objet inconnu x, calculer l'ensemble E des k objets du jeu d'entraînement les plus proches de x (selon une distance à choisir) et attribuer à cet élément la classe majoritaire dans E.

Dans une tâche de classification supervisée, il y a classiquement deux étapes importantes à ne pas confondre :

- La construction d'un modèle à partir du jeu d'entraînement.
- L'utilisation de ce modèle pour classifier un élément inconnu. Cette deuxième étape permet d'estimer l'erreur commise par le modèle en évaluant l'erreur moyenne sur le jeu de test.

Ce qui est un peu perturbant dans la méthode des k plus proches voisins est que la construction d'un modèle selon cette méthode est triviale ; en effet elle consiste juste à stocker la valeur de k, la distance utilisée et le jeu d'entraînement. Ainsi, un modèle construit avec cette méthode sera de type :

```
type modele = {distance : image -> image -> float ;
               k : int;
               donnees : (image * int) array}
```

Pour évaluer la qualité du modèle obtenu, on observe pour chaque élément du jeu de test la classe prédite selon le principe de début de partie. L'algorithme permettant de calculer cette classe est appelé algorithme des k plus proches voisins. On se propose de l'implémenter de manière extrêmement naïve. On choisit d'utiliser la norme 1 comme distance permettant de savoir si deux images sont proches ou non.

8. Ecrire une fonction norme_1 de signature image -> image -> float prenant en entrée deux images et renvoyant la distance qui les sépare selon la norme 1 sous forme d'un flottant.

9. Ecrire une fonction nb_elements de signature int list -> int array prenant une liste ℓ d'éléments de $[\![0, 9]\!]$ et renvoyant un tableau de 10 cases contenant en case i le nombre d'éléments valant i dans ℓ.

10. Ecrire une fonction classe_plus_frequente de signature int list -> int renvoyant l'un de entiers le plus fréquent dans une liste d'éléments de $[\![0, 9]\!]$.

11. Ecrire une fonction construire_modele de signature (image -> image -> float) -> int -> (image*int) array -> modele prenant en entrée une distance, un entier k et un jeu d'entraînement et qui renvoie le modèle correspondant selon la méthode des k plus proches voisins.

12. Ecrire enfin une fonction determiner_classe de signature modele -> image -> int déterminant la classe de l'image en entrée selon le modèle qu'est le premier argument.

13. Construire le modèle pour lequel $k = 2$ et la distance est celle donnée par la norme 1. Déterminer la classe prédite sur quelques exemples de l'ensemble de test à l'aide de la fonction précédente. La classe prédite est-elle toujours celle attendue ?

Partie 3 *Evaluation de modèles*

Afin de déterminer la qualité du modèle obtenu avec $k = 2$, on va le tester sur l'entièreté de l'ensemble de test (plutôt que sur quelques exemples particuliers comme en question 13).

14. Ecrire une fonction pourcentage_erreur de signature modele -> (image*int) array -> float prenant en entrée un modèle m et un jeu de test E et déterminant le pourcentage d'éléments de E qui ont été mal classifiés par m.

Afin de savoir plus précisément sur quel type d'entrée notre modèle commet des erreurs, on calcule plutôt la matrice de confusion du modèle : c'est une matrice de taille 10×10 contenant en case (i, j) le nombre d'exemples correspondant au chiffre i mais qui sont considérés par le modèle comme faisant partie de la classe j.

15. Ecrire une fonction matrice_confusion de signature modele -> (image*int) array -> int array array calculant la matrice de confusion d'un modèle sur un jeu de test.

16. Tester la fonction précédente sur le modèle construit à la question 13 et commenter les résultats obtenus. *Remarque : Ce calcul peut prendre une petite dizaine de minutes.*

17. Calculer le pourcentage d'erreur de différents modèles construits avec la méthode des k plus proches voisins en faisant varier la distance utilisée (ou pourra essayer avec la distance euclidienne par exemple) ou le nombre k de voisins considérés pour déterminer la classe d'une image. On pourra

examiner les images pour lesquelles des erreurs ont été produites et les afficher pour voir si une erreur était prévisible.

Une remarque très importante : le fait que les pourcentages d'erreur varient selon k ne dit **rien** d'autre que ceci : modifier k a une influence sur la qualité du modèle (même chose pour les variations observées en modifiant la distance). On serait tenté de choisir k qui minimise l'erreur sur le jeu de test. On peut le faire mais il ne faut pas affirmer que l'erreur du modèle ainsi choisi sera celle obtenue sur le jeu de test.

L'erreur dans la phrase précédente vient du fait qu'en optimisant k en observant le comportement de nombreux modèles sur le jeu de test on est en fait en train d'utiliser le jeu de test comme un deuxième jeu d'entraînement (un jeu d'entraînement permettant de calculer k). En effet, on est en train de construire un objet, k, à partir de données : le jeu de test. Ce faisant, on optimise k sur un jeu de données connu. Or **pour estimer la qualité d'un modèle, il faut utiliser des données n'ayant pas servi à construire ce modèle**. On n'estime jamais l'erreur faite par un modèle sur les données qui ont permis de l'obtenir.

Malgré tout, il est légitime de chercher à déterminer une "bonne" valeur pour l'hyperparamètre k (les caractéristiques du modèle qui ne sont pas sélectionnées par le jeu d'entraînement s'appellent les hyperparamètres du modèle, par opposition aux paramètres qui eux sont choisis à l'aide du jeu d'entraînement. Pour la méthode des k plus proches voisins, il y a seulement des hyperparamètres et pas de paramètres). Pour ce faire il faut en fait 3 jeux de données : un jeu d'entraînement, un jeu de validation et un jeu de test. Dans notre contexte, pour déterminer un bon k on devrait :

- Construire des modèles $M_1, ..., M_n$ (l'indice i représente la valeur de l'hyperparamètre k choisi pour construire le modèle) à l'aide du même ensemble d'entraînement.
- Calculer le taux d'erreur de chacun de ces modèles sur le jeu de validation. Le meilleur modèle selon ce critère a été généré pour une certaine valeur m pour l'hyperparamètre k.
- Tester le modèle M_m sur le jeu de test.

C'est seulement une fois qu'on aura observé le pourcentage d'erreur de M_m sur le jeu de test qu'on pourra dire si oui ou non m est un bon hyperparamètre.

Le mot de la fin

Afin de bien comprendre pourquoi la méthode des k plus proches voisins est une méthode d'apprentissage un peu atypique, observons le parallèle avec l'algorithme d'interpolation de Lagrange (permettant de traiter la tâche classique consistant à interpoler un ensemble de points par un polynôme). On a :

	méthode des k plus proches voisins	interpolation de Lagrange
hyperparamètres	k, distance $d(.,.)$	degré du polynôme maximal autorisé
construction du modèle	stockage du jeu d'entraînement, k et d	calcul d'un polynôme interpolateur P
prédiction du modèle sur e	algorithme des k plus proches voisins sur e	évaluation de P en e

Autrement dit, la phase de construction du modèle avec la méthode des k plus proches voisins ne nécessite aucun calcul ce qui n'est pas la norme.